TITLE OF THE INVENTION:

Method for generating a simple kind of Artificial Consciousness in a computer, which has the capability to plan, generate automatically and execute machine-code for the solution of arbitrary programming-abandonments. (Automatic Programming)

REFERECES:

First announcement of this utility-patent "Verfahren zur Generierung einer einfachen Form künstlichen Bewußtseins im Computer zur Befähigung selbsttätig planender Erstellung von Maschinencode-Programmen und deren Ausführung zur Lösung beliebiger gestellter Programmier-aufgaben" was the 2nd November 1999 in Germany at the DPMA (deutsches Patent- und Markenamt = german Patent- and Trademark-Office).

SPONSORSHIP STATEMENT:

There was no sponsor for this invention and I'm a single independent inventor.

BACKGROUND OF THE INVENTION:

1. Field of the Invention:

The present invention relates to computer programming, and more particularly to automatic code generation. It relates also to learn-capable programs and artificial intelligence.

2. Present State of the Art:

Worldwide in the Field of Software-Development many employees are missing and the development tasks become larger and larger.
Until now a given conceptual formulation is conceived and programmed by Software-developers.
For relieving the programming there are "Wizards" which offer the possibility to generate basic parts of source-code after making interactive inputs on dialog-windows by a fixed given generator-scheme.
Moreover company-specific scripts are written, which generate simple steadily repeated parts of source-code with variations on the same positions by reading out data out of ASCII-files.

In every case the user first has to develop the generating script and then has to write the ASCII data for to read out, or - in the case of "Wizards" - has to make user defined inputs and after the generation of the frame-sourcecode has to develop the intrinsic functionality of the program.
After it the source-code has to be compiled to become executable. But such programs are not adaptive.

On the area of AI there are neuronal networks / fuzzy logic which can build expert-systems, which can absorb external attractions and have the capability to make adaptive decisions on these inputs, which means a kind of adaptive control system but they will not be able to plan and develop and execute machine-code an learn from its execution.

In the decision 20 W (pat) 12/76 of the german patent court artificial consciousness was tried to generated in a patent application by a reflexive chain of video cameras and monitors - this procedure has nothing to do with that method.

BRIEF SUMMARY OF THE INVENTION:

It's an object of the invention to create computer based artificial consciousness.

It is another object of the present invention to provide a method for giving a computer the capability to learn programming for itself.

It's a further object of the invention to provide a system to make a computer plan and develop programs targeted to a pregiven programming-aim or to fulfill its basic needs.

Therefore it first captures all processor exception-vectors (for a single process-system) or task exception-vectors (for a multitasking version) by own analysis-routines.

Then the system generates numbers, puts them to a defined place in memory and then sets the instruction-pointer to that number for to execute it, like it would be a legal opcode.

Before the number is executed the processor's registers are set to predefined initial conditions and one number is executed several times using different initial conditions.

After every number-execution the system analyses, if an exception occurred or the number caused a jump or a write to memory and the concerned source- and destination-registers are determined and also the kind of instruction, which means its mnemonic. For every execution many theoretical source-registers and several possible commands are possible. Therefore one number is executed by

many predefined different initial conditions to determine the concerned source-register and operation most exactly. By the sum of the execution analysis data the command concerned mnemonic and the source- and destination-registers are determined. So the system itself learns to program in machine-code.

Additional the absolute basic needs, which also have mono-cellars, are modelled:
Pain means an attack to the program (=overwrite) and hunger means loss of energy, which is modelled by a defined register (hunger=low values).

The program has got two valuation-systems: one concerning the basic needs and one concerning the fulfillment of pregiven programming aim.

After single numbers are executed two-number-combinations are executed and the effects of these combinations are determined.

The valuation system determines if the combination is good or worse concerning the basic needs and the fulfillment of a pregiven programming-aim (it's possible to disable one of these two valuation-systems).

The programming-aim concerning valuation-function is dynamic, which means the value-range of its valuation-results is valuated by a meta-valuation-system - and if the valuation-results are not very meaningful, which means, they're clustered near the min/max-boundaries or near zero or another value, the valuation-function is changed by the valuation-system itself and a revaluation occurs. If then the valuations results are worse than before, the modification is quashed and another valuation-function modification is tried until the revaluation results in unclustered valuation-results.

When larger number-combinations are tried, the valuation-system omits to combine combinations which caused fatal exceptions, large jumps, extensive writing to memory, registers which should not be used, etc. or combinations which dislodge from the programming-aim. So not every additional number or combination in the total combination causes an exponential rise of needed calculation-time and disc-space.

The larger the combination becomes the more restrictive is the programming-aim specific valuation-system concerning additional numbers=opcodes or combinations. Then additional combinations must appropriate the programming-aim.

So the system learns to plan developing the desired routine.
The solution-routines are retested by nearly all possible input-values and if it works fine it's

valuated by needed clock cycles and memory space and the most effective solution-routine then is disassembled and can be taken by developers to implement it into their projects as a subroutine.

BRIEF DESCRIPTION OF THE DRAWINGS:

Fig.1 shows the ER-diagram of the database-tables which contain the data of the AC-program. The in the middle shown CLT(i)-tables are created dynamically. The database is described in section 1.3.2.

Fig.2-18 show the names, datatypes, value-range and meanings of the table's columns - some with additional examples, how they're filled. These tables are described in sections 1.3.2.1 to 1.3.2.16.

Fig.19-21 show the value-assignments of the OxT and CxT-tables. Here the effects of the executions are analyzed, and the mnemonic and source+destination-registers are determined.

Fig.22 shows the value-assignments of the energyspecific tables ELT and EBT, which provide the analysis results of the actions concerning the modelled hunger.

Fig.23-24 show the flow-chart of the AC-program.

DETAILED DESCRIPION OF THE INVENTION:

Contents:

1. SPECIFICATION

1.1 Object of the invention:

The objects of the invention are:

a) to provide automatized software-development,

b) to create computerbased artificial consciousness.

With this method a simple form of artificial consciousness is generated using a computer, which acts aimless and arbitrarily in the beginning, but has the capability to learn from the effects of all its "behaviour", for to, when it knows the effects of every single behaviour or behaviour pattern (=combination), has the capability to join together its single actions targeting to a given aim or fulfilment of basic needs.

1.2 Derivation of technical feasibility and Definition of artificial Consciousness:

1.2.1 Philosophical basic liberations:

(... are normally no ingredient of a patent specification, but indispensable for the explanation of the technical feasibility)

If the prerequisite of consciousness of man would be a kind of soul which would be adventitious between cygot and birth, it world be possible to locate it by cogitation experiment:
If you would cognitive cut the head and provide the carotids with oxygen and nutrient containing blood, the consciousness would surely be located in the head.
If you would isolate the brain expect of the factitious supply, no conventionally information flow between the individual and the environment would be possible, but the "I am"-consciousness would surely be present.
Over it it's theoretical now possible to cut the cerebral lappets for seeing, listening, smelling, tasting, feeling, equilibration, speech and the cerebellum and nothing more would be lost.
If the front cerebral lappets would further be cut, you'd loose the possibility to compute with present knowledge and on the cut of several upper cerebral lappets you'd loose reminiscence, but the deepest basic "I am" would be remaining.
⇒ If a kind of soul would exist, it would be located on the upper End of the phylum brain ##.

Under consideration of the fluent evolution the primates would have a "soul" too; and other mammals too; and all other animals too; and the single-cellars too; and plants too; and consequently every cell of a multicellular life form too.
⇒ A border of a "soul-adventitious" or an "I am"-consciousness between the life forms is

missing.

⇒ Every cell of our body ought have its own soul (the evolutionary specialisation to neural-cells is, equivalent to the prenatal cell-fissions, fluent).

⇒ A soul does not exist (it's not necessary to build consciousness).

⇒ Consciousness originates during the evolution compulsively automatically.

⇒ Inside the "dead" molecule of the DNA is the construction plan for generating consciousness.

⇒ Consciousness originates by valuation of the own actions and its effects, with the reflection of the valuation-results on the adapting dynamic valuation-method.

⇒ If no soul is necessary for generating consciousness, but only the complex "program" of DNA, then consciousness is also generatable by a complex reflexive computer-program.

1.2.2 Basic Approach for the Generation of Artificial Consciousness:

The doing of all, including the plainest individuals conduces the fruition of its basic needs.
These basic needs are:

a) no achiness := no attack against the own system *and*
b) no hunger := no imminent loss of energy

A complex program, in which these basic needs are modelled, and which can act freely and has the possibility to learn reflexively what its actions effectuate (like a child) and can reflect if its actions improves its situation in reference to its basic needs, builds up a valuation-system, an then plans with the actions from its learned knowledge and reflects again and so attains consciousness. When it later scans its own machine-code and then tests every of its opcodes and after it all opcode-combinations it discerns the effect of its opcode-combinations and their context and so attains self-consciousness and now has the possibility of self-reproduction and the aware improvement of its machine-code while reproduction and so starts its evolution (its so effective like man could improve its DNA while mitoses/meiosis implementing its experience of life).

1.3. Technical Doctrine how to generate Artificial Consciousness:

1.3.0 Definition of the appropriated abbreviations:

The programming works on every computer with any processor and on any operating-system. In the following μ-indexed abbreviations correlate with Motorola processors, π means Intel processors, and ψ-indexed denote PowerPC-RISC-Processors.

eq. = equivalent

ASR_{ψ} = Adress Space Register

BAT_{ψ} = BAT-Registers

BC = BitCode: every Bit correlates with a Flag and combinations are allowed.

CCR_{μ} = Condition-Code-Register (= Flags: EXtension, Negative-, Zero-, Overflow-, Carry-)

CISC = Complex-InstructionSet Computer (p.e. IA_{π} and MC_{μ})

CPU = Central processing unit = Prozessor

CR_{ψ} = Condition-Register (CR 0..7)

CR_{π} = Control-Register

CTR_{ψ} = Count-Register

$DABR_{\psi}$ = Data Adress Breakpoint Register

DAR_{ψ} = Data Adress Register

DB = DataBase

DEC_{ψ} = Decrement-Register

DR_{π} = Debug-Register

$DSISR_{\psi}$ = DSI Status-Register shows the reason for DSI- and Alignment-Exceptions.

EA = Effective Adress (memory access without using a register)

EAR_{ψ} = External Access Register

$Rseg_{\pi}$ = Segment Register: CS; SS; DS, ES, FS, GS

$EFlags_{\pi}$ = 32-Bit-Register with the System-Flags: IDent-, VirtualInterruptPending-, VirtualInterruptFlag-AlignmentCheck-, Virtual8086Mode-, ResumeFlag-, NestedTask, InputOutputPrivilegeLevel, OverflowFlag, DirectionFlag, InterruptEnableFlag, TrapFlag, SignFlag, ZeroFlag, Auxiliary/Align-CarryFlag, ParityFlag, CarryFlag.

EIP_{π} = Extended Instruction Pointer ($\triangleq PC_{\mu}$)

ER = Entity Relationship (database-model)

ESP_{π} = Extended StackPointer ($\triangleq USP_{\mu}$)

Exc. = *Exception*$_{\pi}$: #DevideError, #DeBug, NMI IRQ, #BreakPoint, #OverFlow, #BoundRange exceeded, #UD (Invalid Opcode), #NM (device not available), #DoubleFault, invalid #TaskSwitch, Segment #NotPresent, #SS (StackFault), #GeneralProtection, #PageFault, #MF (FloatingPoint-Error), #AlignmentCheck, #MachineCheck.

Exception$_{\mu}$: Reset, BusError, AdressError, invalidOpCode, Div/0, CHK, TrapV, PrivilegeViolation, Trace, Interrupts, Traps.

Exception$_{\psi}$: System-Reset, Machine-Check, DSI, ISI, Ext.Interrupt, Alignment, Program, Floating-Point unavailable, Decrementer, System Call, Trace, Floating-Point Assist.

FFT = fast Fourier-Transformation

FK = Foreign Key of a ER-Database-table

FPR_{ψ} = Floating-Point Register 0..31

$FPSCR_{\psi}$ = Floating-Point Status and Control Register

GB = GigaBytes = 2^{30} Bytes

GPR = General Purpose Registers: on Pentium$_\pi$: EAX, EBX, ECX, EDX; ESI, EDI, EBP; ESP; EIP : on PowerPC$_\psi$: GPR 0..31 ; and on Motorola$_\mu$ the Data- and Adress-Registers.

IA = Intel-Architecture

IRQ = Interrupt-Request

AC = Artificial Consciousness

kB = kiloBytes = 2^{10} Bytes

ld = Logarithm dualis = $\log_2$

LR_ψ = Link-Register

MB = MegaBytes = 2^{20} Bytes

MSR_π = Model Specific Register

MSR_ψ = Machine State Register

NMI = Non-Maskable-Interrupt (highest Interrupt)

NOP = NoOperartion-OpCode [=opcode without effect (except increment of IP_π/PC_μ)]

OLB = OpCode Lower Byte = last byte in the opcode

PC_μ = Program-Counter (=Pointer to the first byte in memory where the processor starts to fetch and execute an opcode)

PK = Primary Key of a ER-database-table

PVR_ψ = Processor Version Register

RISC = Reduced-InstructionSet Computer (p.e. PowerPC$_\psi$)

RTE_μ = instruction: return from exception (loads SR and PC from Supervisor-Stack)

$SDR1_\psi$ = SDR1-Register

$SPRG_\psi$ = SPRG 0..3

SR_μ = StatusRegister (Flags$_\mu$: Trace-, Supervisor-, +Interrupt-Maske: I_2 I_1 I_0, +**CCR**-Flags)

SR_π = Segment Registers: CS, DS, SS, ES, FS, GS

SR_ψ = Segment Registers

SRR_ψ = Save/Restore-Register of Machine-Status

SSP_μ = Supervisor-StackPointer (A7 in Supervisor-Modus)

TB_π = Time Base Facility: Time-Counter → 2^{64}-1

TR_π = Table-Register: GDTR, IDTR, LDTR, TR

USP_μ = User-StackPointer (Aderessregister A7 in User-Mode, A7' in Supervisor-Mode)

$\triangleq$ = correlates

$ = begin of a hexadecimal number

ζ = result of bit-by-bit-AND over all following values [= V_1 & V_2 & .... & V_n]

Ξ = result of bit-by-bit-OR over all following values [= V_1 | V_2 | .... | V_n]

Υ = number (sum) of the set Bits in the following value [=(1&V)+(2&V)/2+(4&V)/4+...]

$\forall$ = for all of the following ...

$\forall^-$ = for all other ... (except the following)

1.3.1 Procedure for generating artificial consciousness by comprehensible words:

A computer system attains consciousness, if the active program, in which basic needs are modelled (see 1.3.5), has captured all exception-vectors and proceeds as follows:
Generate a normal number, write it somewhere in the memory, put the program-counter=instruction-pointer on it and execute it like it would be an opcode (=machine-code-command) and analyze, what its execution caused and proceed so with all numbers→opcodes (until a maximum length) with many representative initial conditions (register-values and reference-contents of address-registers).
Then use the saved opcodes which seldom caused an exception while using several initial conditions and evaluate if its execution increased or decreased its situation in due to its basic needs.
Combine the opcodes, which didn't decrease the own situation, and evaluate the effects of the code-combination using several initial conditions and save the result of analysis.
Plan combinations of those opcode-combinations which would increase the well being referable to the basic needs or could fulfill or approximate a given programming aim.

1.3.2 Creating the Database of the AC-Knowledge:

For to have the learned from actions persistent, and for to have convenient access to the large quantity of data, a relational database system with its tables and relations shown in 3.1 is created. For to increase access and to save hard disk capacity, equivalent primary keys in clusters and additional indexes for often used non-PK-rows. The ER diagram is shown in Fig.1.
Processor-dependant and in dependence of the number of 32-Bit-OpCodes, the database can grow very large and so the access speed according to it slow. Therefore RISC-Processors are more applicatively for the AC-Procedure than CISC-Processors. But CISC-Processors, like in the IA, use not very much opcodes which are longer than 24 bit, wherefore it's possible to have with striped tables and additional index-hard-disks and a higher "obliviousness" on inefficient opcode-combinations, an acceptable performance too.
The hard disc space problem in discussed in 1.5.

1.3.2.1 The Register-Identifikation-Table [RIT - Fig.2]:

In the RIT the individual descriptors of the processor-registers and -vectors are typed first: Every Register gets a correlated identification-number, an assigned bit in the BitCode, a character describing the register-type, a consecutive number of the register-type and an optional description of the register. The register of the processor-flags ($EFlags_{\pi}/SR_{\mu}$) gets the Register_ID #0. The exception-vectors mostly are located in memory and are no internal processor-registers - to identify most of these important vectors they get a Register_ID with negative sign, which correlates with

the exception-number. [If exception #0 is not RESET but a real exception, then all exception-vector Register_IDs are displaced by 1: *Register_ID* = -(ExceptionNr + 1).]
Fig.2b shows a Motorola example of the RIT.

1.3.2.2 The Operation-Identification-Table [OIT - Fig.4]:

Like in the RIT the registers, here in the OIT the most important operations get an identification number and a bit in a BitCode.
The *Operation_Type* pigeonholes the operation to an operation group, described in Fig.4c.
The *Operation_Mnemonic* (needn't to be exact like using assembler) and the optional *Operation_Description* describe the basic command.

1.3.2.3 The Initial-Condition-Table [ICT - Fig.3]:

Because of equal opcodes could cause different effects, in this table many representative initial conditions referring to all positive Register_IDs are pregiven here. For every initial condition number (in the fig.3-example: -31..+30) for all positive *Register_IDs* a sample of initial conditions is generated, p.e. using the Function in Fig.3b. But only for all registers, which could contain mathematically used numbers, like data-registers, address-register-references and in the decremented reference of it [to include the command -(adr.reg.)], floating-point-registers and other special calculation-register (like p.e. MMX).
With the content of the address-registers it's surely possible to calculate too, but their values mostly refer to values in memory, where the predefined reference-values of the are located. Therefore the initial conditions of the address-register-values should only gyrate circadian through the references.
The $Status_{\mu}$/$EFlags_{\pi}$-Register higher bytes are always filled with the same initial conditions from SAC.*actual_Processor_Mode*. The ConditionCodes in the lowest byte of $Status_{\mu}$/$EFlags_{\pi}$ have got variable initial conditions. With the Control-, Debug- and Machinestate-Registers and the other special registers is dealt carefully too - the always get the same default-values.

1.3.2.4 The OpCode-Register-Table [ORT - Fig.5]:

For every form initial values by opcode-execution changed (destination-) register, in a loop over all possible source-registers it's ascertained which possible operation-types of the OIT could caused the value in the changed (destination-) register.
For every appropriate source-destination-register-combination a table-entry is generated (unitary operators get the *Register_ID_source* = -1) and for every matching operation-type between possible source and destination the OIT-belonging *Operations_BitCode*-bit is set [p.e. 2 + 2 = 2*2 = 2^2 = 2<<1 = 2|4 = ... = 4 for one or two source registers 2 (the other could be a constant) and one destination-register 4].
The ORT-columns are described in fig.5 and fig.19 contains the value-assignment-algorithms.

1.3.2.5 The OpCode-Learn-Table [OLT - Fig.6]:

The OLT is a subsumption of the effects of the actual opcode on the various used initial conditions. In the first 6 columns informations of fatal effects of opcode-execution are collected. Then there's the difference between the instruction-pointer = program-counter -value after and before opcode-execution and after it the condition-codes which could have caused a jump [redundant to ICT.*Register_Value*(*Register_ID* = 0)].

Then in *Register_changed_BitCode* and *Register_source_BitCode* the Bits of all possible destination- and source-registers follow out of the belonging ORT-entries and after it in *max_Operation_BitCode* and *min_Operation_BitCode* the bitwise *OR*ed and *AND*ed BitCodes of the ORT.*Operations_BitCode* -Entries.

The duration and time of opcode-execution are stored, and in *aim_valuation* analog VFT.*Valuation_Function*(ADT.*aim_Valuation_FunctionID*) it's appraised, how valuable the opcode-execution is in reference to reaching the programming-aim (value-assignment is shown in fig.20).

1.3.2.6 The OpCode-BaseTable [OBT - Fig.7]:

The opcode-base-table shows the ascertained total effect of one opcode in reference to all initial conditions. Fig.21 explains, how the evaluation (column-filling) happens, for so getting a "warrant of apprehension" of the opcode.

The OBT contains the resume of all executions of the opcodes, which later is necessary for the aim-directed planning of code combinations.

1.3.2.7 The Combinations-RegisterTables [CRT(i) - Fig.8]:

The Combinations-RegisterTables are created dynamically, built analog the ORT, with the difference, that effects of opcode-combinations are analysed here. So the CBT(2) has got one more opcode in the primary key than the OBT = CBT(1), and CBT(3) has got three opcodes, etc.

1.3.2.8 The Combinations-LearnTables [CLT(i) - Fig.9]:

Here the same is valid, like in the CRT(i). Analogous the OLT, one CLT(i)-row contains the effects of one opcode-combination in reference of the used initial conditions.

Here first the column *CLT(i).gradient_aim_valuation* gains importance. While it was identical to *aim_valuation* in CLT(1) = OLT, in CLT(i≥2) it contains: *CLT(i).aim_valuation - CLT(i-1).aim_valuation* (see third line of fig.20).

1.3.2.9 The Combinations-Base-Tables [CBT(i) - Fig.10]:

Analogous the CBT(i) show the resume of the effects of all initial conditions in reference to one opcode-combination. The value-assignments are shown in fig.21. CBT(n) (where n is the largest i) is the combination-plan-table - it's the place where the aim-solution-program originates.

If *ADT.aim_fullfilled_Flag_Function(CPT-PK)* = 1 (TRUE), the solution-program of the given aim is found and it'll be enrolled into the AST.

1.3.2.10 *The Aim Solution Table [AST - Fig.11]:*

For every given programming abandonment the found solution-programs, their lengths, execution-times and used registers and operations (→bitcodes) are enrolled here.

1.3.2.11 *The Aim Description Table [ADT - Fig.12]:*

The ADT assigns to every programming aim an identification-number, a short description, one bitcode-combination of the source- and one of the destination-registers which should be used (if possible) and one bitcode-combination of forbidden source- and one of forbidden destination-registers; further a string of former solution-programs which could be implemented, and a aim-solution-flagfunction which returns TRUE if the opcode-combination (program) solves the problem for the desired source- and destination-registers and finally an identifier which references to the valuation_function in the VFT, that appraises the closeness to the complying programming-aim, which is among others dependant of this aim-solution-flagfunction.

1.3.2.12 *The Function-Identification-Table [FIT - Fig.13,14]:*

In the FIT basic subfunctions are provided, which can be used for composing the energy valuation-function.

It's introduced in two variations:

a.) for generating a dynamical valuation function in SQL,

b.) for generating a dynamical valuation function in machine-code.

The alterable building of a valuation function is easier to accomplish in SQL, but the execution-time in machine-code is much faster and every new composed SQL-valuation-function has to be parsed again.

In the future the valuation-function should only be composed in machine-code. This has the additional advantage that the AC-program could use some solved solution-functions again as subfunctions in the FIT for later use for composing the valuation-function.

1.3.2.13 *The Valuation-Function-Table [VFT - Fig.15]:*

The VFT contains the dynamic valuation-system in reference to the own "well being" (energy-register) and to the closeness to the programming aim(s).

The VFT.*Valuation_Function*(Type='E', SAC.Energy_Valuation_Function_ID) appraises energyspecific actions and the *Valuation_Function*(Type='A', SAC.Aim_Valuation_FunctionID) the closeness to the programming aim(s).

The VFT.*Function_ID_Chain* contains the concatenation of the FIT.Function_ID's, that means the execution-chain of the subfunctions: Here causes NUM (see fig.13b), that the following value is used as a number of byte-length, VALUE denotes that the following value is the column-number of the CPT=CLT(n), from which actual row the value is taken, EREG means the Register_ID of the energy-register, S/D_REG denotes the value out of ADT.all_source/dest_Registers_BitCode and AIM_F is the

result of the ADT.aim_fulfilled_Flag_Function. The unitary operations operate on the last result of the Function_ID_Chain and the binary operations on the last two results.
On every accommodation, enhancement, or other amelioration of these valuation-functions the *Valuation_Function_ID* is incremented and a new entry with the modified *Valuation_Function* is created and the efficiency of all valuation-functions are reappraised:
VFT.*Valuation_Function_value* = SAC.*Energy/Aim_self_valuation_Func(...)*], for to have an efficiency-gradient for further improvements.
The functionality of the dynamic valuation-system is described in 1.3.7.

1.3.2.14 *The Status of the AC-Program [SAC - Fig.16]:*

This table has no primary key and only one row. It contains status-informations of the AC-Program and two self-valuation-functions, which appraise the efficiency of the energy-valuation-function and of the valuation-function of the programming-aim-closeness (VFT) by evaluating the range of their valuation-results.
These self-valuation-functions are, in opposite to the energy- and aim-closeness valuation-functions, not modifiable by the AC-program itself, but can be changed by the user.

1.3.2.15 *The Energy-Learn-Table [ELT - Fig.17]:*

In the ELT data are stored about all energy relevant actions over the actual initial conditions, that means for all opcodes and code-combinations, which pertain the last data-register.
The valuability of an energyspecific action is appraised according to ELT.*Energy_valuation* = VFT.*Valuation_Function*(SAC.*Energy_Valuation_Func_ID*).

1.3.2.16 *The Energy-Base-Table [EBT - Fig.18]:*

Like in the CBT(i) for programming-aim-closeness, in the EBT the effects of energy-changing opcode-combinations for all initial conditions are collected.

1.3.3 Preparing the initial State of the System

For to reset the system later into the initial state without booting, several pointers have to be latched. Afterwards all exception-vectors are intercepted by own routines, because of the initial trying to use arbitrary numbers as machine-opcodes, although many of these trying cause fatal exceptions, because they're illegal opcodes (not usable) or the opcode causes an exception on one of the initial conditions. Abnormal system end would be the consequence, if not all exception-vectors would be captured.

In the case that the AC-program should run preclusively, you'll have to
a.) stop multitasking by disabling it by an operating-system routine or by setting the IRQ-mask of

the processor to NMI.

b.) save all system-exception-vectors.

c.) set all system-exception-vectors to own analysing and handling routines.

or if it should later run with other programs or perhaps with further AC-programs:

a') increase own task-priority.

b') save all task-exception-vectors.

c') set the task-exception-vectors of the AC-program to own analysing and handling routines.

---

d.) save the statusregister$_{\mu}$ ($\triangleq$EFlags$_{\pi}$) and the user-stackpointer.

e.) save the values of the other address-registers and of the data-registers.

f.) save the values of the segment-, control-, debug- and special-registers.

g.) set exception-vectors, which load additional data to the supervisor-stack (p.e. on address-violation-exception several processors load additional information like access-address and opcode onto the supervisor-stack) to a this fact considering interceptor-routine.

h.) set the privilege-violation–exception-vector to a special capturing routine.

i.) set one trap-vector to a routine, where the system should continue inside the supervisor-mode after this trap occurs.

j.) execute this trap intentionally to change the CPU-mode from user-mode to supervisor-mode (the system now continues on the above set routine).

k.) set the trace-exception-vector to an own trace-routine for later effect-analysis after number-as-opcode execution.

l.) set the bits in the first word on the supervisor-stack so, that when the SR is loaded from SSP, the trace-Flag is set and the IRQ-mask is set to NMI (p.e. this is #$8700 on Motorola) because while the following base-opcode-learning no interrupt should be possible and after execution the effects should be analysed.

See referring to this fig.24a.

1.3.4 Base-learning from execution of all single opcodes:

1.3.4.1 OpCode generation and execution:

a.) Generate a 32-Bit-Number as an opcode, starting on #$0000.0000, later increase by 1. [if you already know the CPU instruction set, you can skip the opcodes which would overwrite memory (p.e. memory-move-commands).]

b.) Set data- and address-registers, and the address-register destination-values and the values one DWord below on predefined test-values (initial conditions) and clear the condition-codes in EFlags$_{\pi}$/CR$_{\mu\psi}$ (or use several initial conditions too).

c.) Write the above generated opcode into the test-location in memory. Then fill the memory behind with zeros up to the maximum possible opcode-length, if zeros mean the mnemonic

"ORI #0, Reg.0" (or another effectless command) or fill with NOPs.

This is necessary because on a long command the zeros are not meaningless [and then often less destroying (p.e. memory overwriting) than the NOP-corresponding opcode-number], and on a few processors a clearing of the trace-flag is possible in the while execution used user-mode too, which effects the execution of the following numbers as opcodes too [if now the zeros are not effectless, NOPs have to be used to prevent further effects (like memory- or program-selfoverwriting)].

After these zeros or NOPs the Trace-Bit-Cleared handler is following.

d.) Set content of the supervisor-stack, which is loaded by return from supervisor-mode into important user-registers like EFlags$_\pi$/Status-Register$_\mu$, IP$_\pi$/PC$_\mu$, etc., so, that CCR will be cleared or set on initial conditions, IRQ mask will be set to NMI, the trace-bit will be set and the supervisor-bit[mask] will be cleared and the DWord behind is the location of the test-opcode.

Now execute the return from supervisor-mode by the corresponding opcode-command (p.e. RTE$_\mu$): EFlags$_\pi$/Status-Register$_\mu$ is now loaded by above described values and the test-opcode executes, because IP$_\pi$/PC$_\mu$ is loaded by its address.

- If now a fatal exception occurs (except trace, p.e. privilege violation, etc.), the kind of exception is briefly shown graphically if desired, and it will be continued by generating and executing the next opcode.
- If an initial condition dependant exception occurs (like address error, division by zero, ...), a relation between initial condition an exception is analysed [attention: on several exceptions, because of trace, an in many literature not documented combination of both exceptions (internal processor handling) can occur (p.e. using M68000 on Trap, Chk, Div/0 in combination with Trace).]
- If no exception occurs (not even trace) the opcode-execution cleared the trace-bit (should never occur while executing single opcodes) and the handler behind the opcode is executed.
- On Trace-Exception (normal case) a usable opcode was generated which execution-effects have to be analysed now.

1.3.4.2 Analysis of opcode-repercussion and saving the analysis-result:

a.) After execution the EFlags$_\pi$/Status-Register$_\mu$ and the data- and address-registers and the reference-values of the address-registers an the reference-values one DWord below an the user-stackpointer are saved for analysis.

b.) Verifying the own machine-code-checksum (of AC-Prg.) and the inactive copy in RAM (both without test-opcode placement): If checksum changed, the AC-program injured itself while executing the test-opcode (overwrote own parts of program). The corresponding corrupt-flag is set in the table. If the active version checksum changed jump into the inactive version, then compare both versions byte by byte and repair the corrupt version by replacing the bytes in the version with the changed checksum by the bytes from the version with the correct checksum.

c.) Check the supervisor-bitmask of the saved user-stackpointer on the supervisor-stack: If the

processor was in supervisor-mode before jumping into the exception-handler, although he was in user-mode during test-opcode execution, a combination of the normal trace-exception with a low-priority-exception occurred [p.e. on Motorola Div/0-, Trap- or Chk-Exception (not documented in the M68000er handbook)]. That means the processor first fetched the exception-vector of the just occurred low-priority-exception and loaded the statusregister and the program-counter onto the supervisor-stack and then he jumped, while being in this processor intern routine, before starting the corresponding handler of the exception-vector, into the trace-exception-routine, which caused a further loading of the program-counter and the status-register (where now the supervisor-bit(mask) is set) onto the supervisor-stack.

By analysing the supervisor-bit(mask) on the supervisor-stack, it's now detectable, that before trace another low-priority-exception occurred; and by comparison of the second saved program-counter below on the supervisor-stack with the low-priority exception-vectors, now the primary exception before trace is detectable, which corresponding exception-number is stored.

d.) Comparison of the IP_π/PC_μ on the supervisor-stack with the address (placement) of the test-opcode-address: If the IP_ψ/PC_μ decreased, stayed unchanged, or increased by a value, which is larger than the longest possible opcode, the test-opcode was a jump-command.

If the IP_ψ/PC_μ increased by 5 or more bytes and less than the longest possible opcode, it was a long opcode or a short forward jump. If no registers changed from initial conditions it was a short jump.

This analysis-result is stored too.

e.) Comparison of the $EFlags_\pi$/Status-$Register_\mu$ and of all register-values and of the destination-values and of the destination-values one max. address-length below [because of -(Adr.Reg.)], with the original-values.

In a bitmask now it's flagged which registers or its destination-values changed and it's analysed, which operations on which source- and destination-registers could have happened (heeding changes of $EFlags_\pi/SR_\mu$) and the result is stored in the ORT and OLT (see figs.5,19; 6,20) and the OBT is actualised (fig.7,21).

f.) If it was a jump-command, in the $EFlags_\pi/SR_\mu$ it's analysed if it was an conditional jump.

1.3.5 Realisation of the basic needs:

1.3.5.1 Realisation of artificial pain:

Pain is caused by system-injuring. The basic pain in the biological evolution incidences already on monocellars and is the injuring of the DNA in the cell nucleus. If the DNA is damaged, the monocellar has to take time to repair its DNA using its resources. It does it by filling missing aminoacids in the defect half of the double helix by adding the complement aminoacids to the undamaged half.

The AC program is loaded twice into RAM. If the AC program (or another one) executes a

command which overwrites a part of the active or inactive AC program, which means an injuring of the active or inactive code (DNA), it has the ability to recognise the damage by comparing the checksum, and has now to take time to repair the damaged code by comparing damaged and undamaged code to have information about the damage-location (valuable information for test-opcode analysis) and then copying the code from the undamaged version into the injured version to heal itself. If the active code was the injured one (had the corrupt checksum), it first has to jump into the inactive duplicate to prevent errors on self-healing, because the healing-routine itself could be damaged.

1.3.5.2 Realisation of artificial hunger:

Hunger means imminent loss of energy. Energy is engendered in the cells by transforming adenosintriphosphat to adenosindiphosphat. The energy for building up adenosintriphosphat from adenosindiphosphat is gained by combustion of glucose. Missing energy (ATP) makes metabolism and with it every action, reaction on pain, or self healing on injury, impossible.

The "energy quantity" of the AC-program is modelable by the height of a value in a data register. Now it would be possible to realise hunger by decreasing the electric current to the processor by external reading of this data register and increasing an ohm-resistance inverse proportional to the register value.

A less authentical hardware-unbound solution is possible too:

Less energy is harmful for learn-process. Frugal values in the energyspecific data-register cause lower functionality on learning from opcode-executions. On less values no learning from opcode-execution is possible. And lowest values cause loss of the saved knowledge from earlier opcode-executions. If the value is zero, additional pain, which means own code injury, occurs.

On hunger the AC-program consequently has to find and execute opcodes, which increase the value of the energyspecific data-register.

Decreasing energy, which means the origin of hunger, is simulated by decreasing the energyspecific data-register by 1 after every action (=opcode-execution) [p.e. by the AC-program itself].

1.3.6 Planning on the criterions of the valuation-system:

If the system tested all possible opcodes and analysed the effects of the suitable commands, it has now the possibility to learn planning aimed to comply its basic needs or its programming-aims: Therefore it combines the opcodes, executes them using all initial conditions and analyses, what effected the code-combination on every single initial condition.

Because mostly longer opcode-combinations are necessary to fulfill the programming-aim, it plans the code combination by using only codes which caused no injury (own damage) or better no RAM access at all and caused no fatal exceptions (divide-error or overflow-exception are allowed) and

used no forbidden registers or opcodes (ADT.*unused_Registers_BitCode*| ADT.*unused_Operations-_BitCode*). OpCodes which use the desired destination and source-registers are preferred (ADT.*all_source/dest_Registers_BitCode*).

ADT.*aim_fulfill_valuation_mode* appoints, if the valuation-function is existent in SQL or directly in machine-code. For the beginning user the slower SQL-version is more convenient and the specialist would prefer to use an assembler–*aim_fulfilled_Flag_Function* (ADT) which is transformed into machine-code, because it's much faster on complex valuation-functions than SQL.

1.3.7 The dynamic-reflexive valuation-system:

1.3.7.1 Valuation of the programming-aim closeness:

The ADT.*aim_fullfilled_Flag_Function*(Aim_ID), returns TRUE, if the programming-aim is obtained and the VFT.*Valuation_Function*(Type='A', ADT.*aim_fullfilled_Flag_Function*, VFT.*Function_ID-_Chain*), supplies a signed-byte value, which means the closeness of the actual CLT(n)-opcode-combination on the used initial conditions to the aim-solution. The result is stored into CLT(n).*aim_valuation* and builds up in comparison with the last CLT(n-1).*aim_valuation* the gradient CLT(n).*gradient_aim_valuation.*

Because of the solution program has to work for all initial conditions, the maximum- and the average valuability of the opcode-combination, both as average over all initial conditions, is stored in CBT(n).*max_aim_valuation* and CBT(n).*avg_aim_valuation* ; and the gradients to the corresponding values of the last CBT(n-1) build up CBT(n).*max_grad_aim_valuation* and CBT(n).*avg_grad_aim_valuation*.

If the boundary-value of -128 or +127 was the valuation-result, the VFT.*boundary_value_counter* is incremented, and analog *low_value_counter* is incremented, if a valuation-result between -16 and +15 occurred.

Using these statistical data, and on an analysis of all CLT(i).*aim_valuation* -values, p.e. if you count the number of values in a small value-range running from min-possible-result to max-possible-result (-127 to +128) in dependence of the width of the value-range-window, the SAC.*Aim_Self_Valuation_Func* appraises after every programming-aim attainment the valuation-results of the VFT.*Valuation_Function* and with it its efficiency.

If the most valuation-results of the VFT.*Valuation_Function* p.e. were near the boundary-values (min./max.), the valuation-function was too steep and has to be flattened, which means inside the VFT.*Function_ID_Chain* have to be more elements with negative FIT.*Function_Flatten*. The opposite is valid, if the most valuation-results caused a high VFT.*low_value_counter*.

So after every solution of a programming-aim a self-valuation of the valuation-function occurs and a further step in the self-programming of the valuation-function. New elements are added to the valuation-function and sometimes elements are omitted and the steepness is adapted.

Then the valuation is done again and it's revised if the new valuation-function would have supplied

a better range of valuation-results.

If the new range of valuation-results was worse than the one before (valued by SAC.*Aim_Self_Valuation_Function*) then the modification of the valuation-function is quashed and another modification is tried. If the changing of the valuation-function ameliorated the range of valuation-results and the self-valuation-function returns a positive value, then it's continued with the next programming-task - otherwise a further amelioration of the valuation-function has to be done until self-valuation returns a positive value.

1.3.7.2 The dynamic energy-valuation-function:

The dynamic energyspecific valuation-system runs as follows:

0.) Because of the results of the energyspecific valuation-system are constricted to the range of *signed_byte* the valuation-function is embedded into a frame:
valuation-result := MIN[MAX(*valuation-function*, -128), +127)]

1.) The energy-valuation-function of 0-th order is "how saturated am I after the action ?":
valuation-function(0) := MIN[MAX(*Energy_after*, -128), +127)]

2.) The valuation-function of 1st order is "how much more saturated am I after the action than before ?":
valuation-function(1) := MIN[MAX(*Energy_after – Energy_before*, -128), +127]

3.) Because of the energy-register is of the type *unsigned integer* (DWord), the boundaries of the valuation would too often the result. Therefore a kind of logarithm or ...
valuation-function(2) := MIN[MAX[SQRT(*Energy_after – Energy_before*), -128], +127]

4.) Now negative energy-gradients would cause wrong signs, therefore 3rd square or ...
valuation-function(3) := MIN[MAX[SGN(*EnergyGrad*)*SQRT(*EnergyGrad*), -128], +127],
where *EnergyGrad* = *Energy_after – Energy_before*

Possibly the function ½·SGN(*EnergyGrad*)·SQRT(SQRT(*EnergyGrad*)) would be better, because it reaches exactly to the boundary-values, but maybe the boundary-values are reached very seldom and a subtler structuring around zero would be more important.

This depends how often the boundaries are reached and how much energy-gradients supply small values. Maybe the naked result-value (*Energy_after*) has to be weighted stronger and a valuation of the gradient alone is not sufficient. Moreover it had to be considered how much and which further registers are concerned beneath the energy-register and which and how much types of operations were executed, etc., and finally the execution-time of the energy-valuation-function itself. Therefore the energyspecific valuation-system has to be rarefied and adapted (like the valuation-system of intelligent biological life forms).

Using dynamic embedded [PL/]SQL the changing and reparsing of the as string stored valuation-function no problem. Because of the execution-speed and the possibility of implementation of earlier programming-aim solutions the energy-valuation-function should be used in machine-code prospectively.

The task of the amelioration of the energyspecific valuation-function is done, like the programming-

aim specific one, after every fulfilling of a programming-aim.

Valuation-system and valuation-results are always reflexively.

1.3.8 Reaching Self-Consciousness, Reproduction and Evolution:

Through the process of self-healing on pain/damage the program knows its location in memory. It now has the possibility to check out the effects of its own opcodes one after another, then consecutive opcode-combinations etc., and when it finally knows the effect of its total length, it'll reach self-consciousness and then has the possibility to reproduce itself and to ameliorate itself awarely while reproduction using its acquired knowledge (p.e. by removing the incommodious decrementing of the energy-register).
The intelligent conscious-varying reproduction is much more preeminenced than the biologic-genetic one, because the latter only refers to existing genes/DNA, while the AC-program can vary itself by changing and extending its code intentionally using its "experience of life".

1.4. Conceptual Formulation of the Programming-Aim and Examples of Achievement

To the AC-program an arbitrary programming-aim is challenged by giving one or more criterions in ADT.*aim_fulfilled_Flag_Function*, by which it can check out, if it solved the task.
Its job is it now to develop a program, which solves the problem for all initial-conditions.

1.4.1 Example_1: Developing a Program to compute the average:

A very simple but easy to comprehend job for the AC-program could be: "write a program that computes the average over two integer-variables".
The AC-program fulfills this task, if the difference between the result and the lower number is equal to the difference of the higher number and the result, and this is functioning for any arbitrary input-numbers.
But the AC program doesn't know the instruction-set of the processor - it knows now only the opcodes which caused no damage and no fatal exception and it knows the opcode-effects in reference to the different initial-conditions.
Through corruption-selfhealing or the energy-register it already knows easiest abandonments like "execute an action which causes no pain" or "execute an action which makes me saturated".
For attaining economic programming-aims, it now needs valuation-variables, which reveals answers to the following questions:

a.) How much nearer or farther away from the programming-aim took me the last added opcode-combination (that every added single opcode of it may cause opposite effects is irrelevant).

b.) How many processor clock-cycles needed the solution-program.

c.) How many bytes long is my program and how many opcodes are included ?

These answering variables (CBT-table-columns) are:

aim_valuation ; cycles_of_execution ; OpCode_length_or_jump.

The input-variables in the example-task could be in the first two data-registers (EAX_π, EBX_π respectively $D0_\mu$,$D1_\mu$ respectively $GPR0_\psi$, $GPR1_\psi$), here *R0* and *R1*.

The return-value should be the third data-register ($ECX_\pi|D2_\mu|GPR2_\psi$), here *R2*.

If the task is solved for arbitrary input-values, the program is ready, because it's function.

If there are more than one solution, the one is chosen which needs less clock-cycles.

The task-specific aim-fulfilled valuation-function, which computes OLT.*aim_valuation* is consequently in this example:

ADT.*aim_fulfilled_Flag_Function*(average of R0 and R1) = { (R2-R0)=(R1-R2) }

Here the problem could occur, that one input-value is even and the other one is odd, which means that this input-combination would have no solution. To implement that, the *aim_fulfilled_Flag-_Function* should be enhanced for integer-variables: { (R2-R0)=(R1-R2) || (R2-R0)+1 =(R1-R2) }.

The AC-program will find several solution-programs and takes the one with the least needed clock-cycles.

A possible solution would be in CBT(3): MOV R0,R2 ; ADD R1,R2 ; SHR R2

(... naturally in machine-code of the used processor - using a Pentium that would be the 48-bit-number #$89C2.01CA.D1EA, using a Motorola that would be #$2400.D282.E2C2 and using a PowerPC (RISC) a 96-bit-number would be the solution).

1.4.2 Example_2: generation of a programs for computation of the cube-root:

A further easy development-task would be "write a program that returns the cube-root of a FFP (fast floating point) number"; the input-variable should be R0 (EAX_π) and the output-variable R3 (EBX_π).

The AC-program fulfills this task, if the result multiplied with its square is equal to the input-value (and this is valid for all initial conditions):

⇒ *aim_fulfilled_Flag_Function(cube root)* = { (R3*R3*R3)=R0 } (←naturally in FFP-multiplic.)

A single command like the square-root (FSQRT) doesn't exist for the cube root.

The solution-program could be in CBT(8) using a Pentium II as follows:

```
Op1(16b):  MOV   CL,3          ;ECX=$????:0003          [1011.0001:0000.0011]
Op2(16b):  FLD1                ;ST(0)=1.0               [1101.1001:1110.1000]
```

Op3(16b): FIDIV CX ;ST(0) = $^1/_3$ [1101.1110:1111.0001]
Op4(16b): FLD EAX ;ST(0) = R0 ;ST(1) = $^1/_3$ [1101.1001:1100.0000]
Op5(16b): FYL2X ;ST(0) = $^1/_3 \log_2(R0)$ [1101.1001:1111.0001]
Op6(16b): FLD1 ;ST(0) = 1.0 ;ST(1) = $^1/_3 \log_2(R0)$ [1101.1001:1110.1000]
Op7(16b): FSCALE ;ST(0) = 1.0*2^[$^1/_3 \log_2(R0)$] [1101.1001:1111.1101]
Op8(16b): FST EBX ;EBX = 2^[$^1/_3 \log_2(R0)$] [1101.1001:1101.1011]

(... naturally only the second of these columns as a 128-bit-number with the bits set as shown in the last column.)

Hexadecimal that would be: B103.D9E8.DEF1.D9C0:D9F1.D9E8.D9FD.D9DB.

This would be a possible solution-number (=program) for the given task (there're surely shorter and faster solutions too).

1.5. Needed Hard-disk-Space and Oblivion

In both examples 16-bit-opcodes would be sufficient, but it's obvious, that large programming-aims would need much hard-disk space. Therefore the AC-program has to forget unimportant or error-causing opcode-combinations.

1.5.1 Table sizes:

IST, RIT and CIT need neglectable disk-space.

Theoretically there could be *size*(OBT) = $2^{32} * \sum$ bytes(column(i)) = 485 GB, but also on a RISC processor never all 32-bit-combinations are used as a valid opcode and realistic are as an average on RISC processors about 28 bits ⇒ 30 GB and on CISC processors about 20 bits ⇒ 118 MB [on latter the most are 16 bit-opcodes, there're a few 8-Bit- and several 24- and 32-bit-opcodes, and the ones which are longer 32-bits are not used (we don't need memory-to-memory-operations for example and the functionality of one long opcode can be substituted by two or more shorter opcodes).

The 62 initial conditions can cause *size*(OLT) = $2^{[20..28]} * 62 * \sum$ bytes(column(i)) = 3 GB (CISC) up to 832 GB (RISC) and *size*(ORT) could reach the same quantity, considering that one opcode mostly pertains one destination- and one source-register (unitary ones use only the destination-register and a few seldom opcodes use 3 or more registers). But there could be many effect-belonging *Operation_BitCodes*, which would increase the tablespace dramatically, if it wouldn't be compensated by the many opcodes which cause an exception, where less information has to be stored.

A much larger problem is the exponential growth of the CxT(i), because every i multiplies the needed tablespace by factor of $2^{[20..28]}$. But this is exact that what should be compensated by the dynamic valuation-system. It decreases its knowledge-absorption tolerance referable to the

remaining hard-disk space: So opcode-combinations with least CBT(i).*max_aim_valuation* or *.avg_aim_valuation* are forgotten and so will not be combined with other opcodes.
Although if the demand of hard-disk space arrears large, this is no problem in near future. Also the according to the combination-possibilities and table-sizes increasing calculation-times are compensated by larger and faster becoming hard-disks and the increasing efficiency of the processors.

1.5.2 Oblivion:

Like all intelligent life-forms, the system has to forget unimportant and less important informations, because
a.) the disk space is limited, and
b.) data access time becomes slow on very large data-tables.
Therefore after every satisfactorily achievement of objectives, when a new programming-aim is given, the ELT and all CxT(i)-tables over a remaining disk-space dependant i are deleted, and the opcode-combinations in the remaining CxT(i) are revalued referable to the new programming-aim and forgotten opcode-combinations are added, if they're valuable for the new aim and the higher CxT(i) are recreated dynamically.

1.6. Becoming Conscious

Through try_and_error the program learned what effectuated every action and what are the effects of which sequence of actions.
Through corruption-healing (if own code was overwritten in RAM) it had to repair its code and so it knows its position in memory.
If it once knows the effect of its own machine-code, it gets self-consciousness and has the ability to reproduce its code and to ameliorate it while reproduction.
Through the so initialised evolution the AC becomes complexer and better and will be able to solve larger and larger programming-tasks in future.

1.7. Presentment of the Economic Advantages

Here a totally new area of using a computer is presented. While normally in a computer run by man generated programs, which execute user-controlled applications, the AC-program itself develops and executes programming-aim oriented routines, which can later embedded into a large application.

The demand for software-development is worldwide much larger than the human potential of developers.
A system which learns to write programs itself has the capability to solve smaller development-tasks, and will in future, after several evolution-steps, have the capability to develop complex programs as the solution of large tasks too, if it has got enough hard-disk space.

The programmers will not have to develop all the routines they need - they order the routine from the AC-program and embed it into their application. So the companies can finish and sell their software-products earlier.